IN-33
148124
P.22

NASA Contractor Report 189730

A Comparison of Multiprocessor Scheduling Methods for Iterative Data Flow Architectures

Matthew Storch

University of Illinois
Urbana, Illinois

Grant NAG1-613
February 1993

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-0001

TABLE OF CONTENTS

LIST OF FIGURES

A Comparison of Multiprocessor Scheduling Methods for Iterative Data Flow Architectures

1. Introduction

1.1. Purpose

This paper provides a qualitative comparison between the Algorithm To Architecture Mapping Model (ATAMM) [1] and three related works [2, 5, 7], with the primary focus on [2]. The problem domain is non-preemptive scheduling of iterative, large-grain data flow graphs as may typically be found in signal processing applications. The purpose of this paper is fourfold: to resolve differences in terminology used by the various authors, to highlight the similarities and differences between ATAMM and the other models, to point out the relative features and limitations of the various approaches, and to suggest possible directions for future ATAMM research.

1.2. Assumptions

- All schedules in this paper are assumed to be multiprocessor schedules, so *schedule* will be used as shorthand for *multiprocessor schedule*.
- Unless otherwise stated all observations regarding ATAMM behavior are for a graph running in steady-state.

2. Terminology

2.1. ATAMM Terminology

In order to facilitate a comparative discussion of the ATAMM and Optimum Unfolding scheduling strategies, the terminology used in the two strategies will be introduced and contrasted in this section. Some familiarity with both ATAMM and [2] is assumed; the terms used in both works are introduced not to give precise definitions but rather to provide a mapping between the two models. The terminology used by ATAMM will be covered first, and then it will be shown how the definitions used in [2] relate.

A data flow graph in ATAMM is called an *algorithm marked graph* (AMG). Two other graph types are used in the ATAMM design system, namely the *node marked graph* (NMG) and the *computational marked graph* (CMG). While these graphs are fundamental to the ATAMM model and ATAMM design procedure, they are primarily a means by which data packet injection interval and node firings are controlled. The effects of the NMG and CMG, such as controlling TBO, ensuring that buffers are not overwritten, and ensuring steady-state operation, are important but the exact manner in which the NMG and CMG are used to achieve those effects need not be reviewed to compare ATAMM schedules with [2], [5], and [7].

An AMG consists of *nodes* representing large-grain computations and directed *edges* from one node to another (not necessarily distinct) node, which represent the flow of data and thus indicate temporal precedence constraints. A *source* is a special type of node that has no incoming edges but nonetheless produces tokens at a fixed rate. A *sink* is also a special type of node that has no outgoing edges, and which therefore produces no token when it fires. One or more *initial tokens* indicating the presence of initial data may be placed on any edge before the graph begins execution. The successor node of an edge will use the nth token from the predecessor to produce the (n+d)th token of the successor, where d is the number of initial tokens. The nth *packet* of data consists of the nth token produced by each node excluding the sinks, which never produce any tokens.

The time between the completion of two consecutive packets is called *TBO* (time between outputs). Part of a design procedure may be to achieve a certain desired or *target TBO*. When a graph is either simulated or run on actual hardware, the *actual TBO* may slightly vary or even oscillate if injection is not controlled properly. Since this paper is primarily concerned with operation at a theoretically perfect steady-state, from here on TBO should be read as target TBO unless otherwise stated. In an injection-controlled environment such as ATAMM, TBO is necessarily equal to the injection interval for the graph to run in steady-state. The smallest achievable TBO for any number of processors is called TBO_{ALB} where the subscript indicates absolute lower bound. In [1], TBO_{ALB} is shown to be the maximum time per token of any directed circuit in the AMG. Let C_i be the ith directed circuit (numbered arbitrarily), $T(C_i)$ be the sum of execution times of the nodes in C_i and $M(C_i)$ be the number of initial tokens in C_i, then

$$\mathbf{TBO_{ALB}} = \mathbf{Max}\left\{\frac{\mathbf{T(C_i)}}{\mathbf{M(C_i)}}\right\}. \qquad (1)$$

The smallest achievable TBO for a given number of processors is TBO_{LB}. Let TCE be the sum of all node execution times and R be the number of processors; then

$$\mathbf{TBO_{LB}} = \mathbf{Max}\left\{\mathbf{TBO_{ALB}}, \frac{\mathbf{TCE}}{\mathbf{R}}\right\}. \qquad (2)$$

If there is no directed circuit (recurrent loop) in the AMG, then $T(C_i) = 0$ and $TBO_{ALB} = 0$. There is another factor which may limit TBO_{ALB}. If it is assumed that nodes cannot be multiply instantiated[1], then TBO_{ALB} will be either the result of Equation 1 or the largest node execution time, whichever is larger. Such an assumption is made in [5]. However, ATAMM [3] directly allows for multiple instantiations, and a similar effect is achieved in [2] by *unfolding* the DFG. Unfolding is a transform that takes a data flow graph G and an *unfolding factor J* as input and constructs a new graph which contains J copies of each node of G. The J copies of a node A of G correspond to J consecutive instantiations of A, and edges are added to the unfolded graph so as to enforce all of the intra-packet and inter-packet data dependencies. See [2] for the unfolding algorithm and additional discussion.

The execution of a node on some data packet is referred to as an *instantiation* of that node. A node is said to be *multiply instantiated* if there exists an instant of time in which the node code is operating on two distinct data packets simultaneously. Although *instantiation* is primarily a software term, it can apply to hardware as well. Note that for our purposes a pipelined hardware multiplier which is in the process of computing several results in its different pipeline stages is in essence multiply instantiated. Two even more closely related hardware examples are a multiple-issue CPU with multiple identical functional units, and a supercomputer with multiple vector units on each CPU.

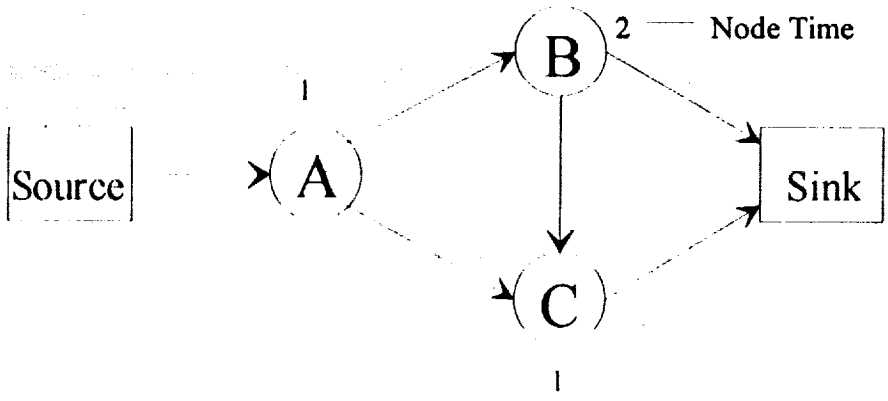
A schedule is *periodic with period TBO* if and only if the following property is met. If a node fires at time t, then the next time that the node will fire is exactly t+TBO. Both ATAMM and [2] are concerned only with graphs which operate in a periodic fashion. If a graph executes in a periodic fashion it is said to have reached *steady-state*. There may be a length of time when a graph first begins executing that it is not in steady-state, in which case a *transient condition* exists.

A *total graph play diagram*, or TGP, is a graphical snapshot showing the node activity at steady-state for exactly one TBO interval of time[2]. The earliest-created packet is usually numbered 1, the next earliest is numbered 2, and so forth. An example AMG is shown in Figure 1(a), a feasible schedule is given as Figure 1(b), and the resulting TGP is shown in Figure 1(c).

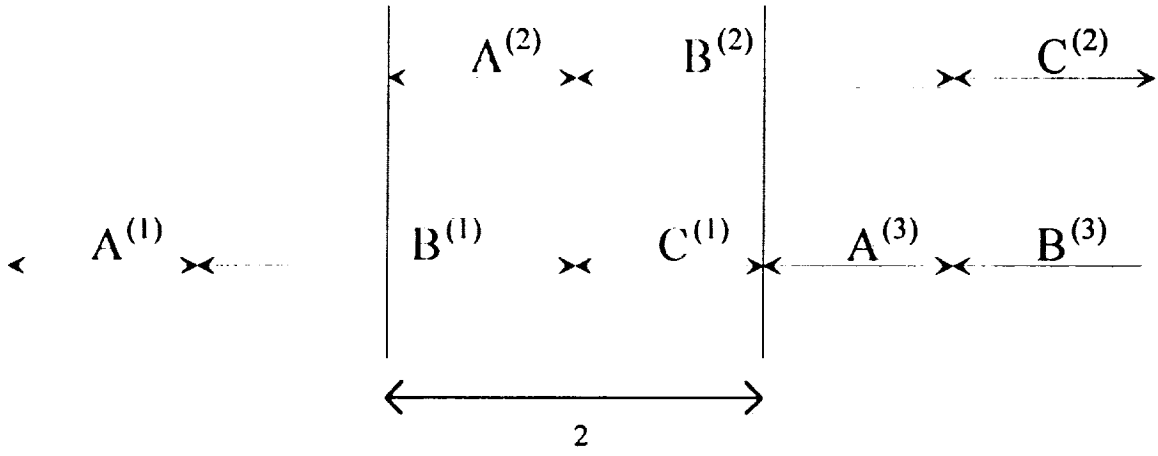
For a complete introduction to ATAMM and a preliminary system implementation see [3].

[1]See next paragraph for a definition of multiply instantiated.

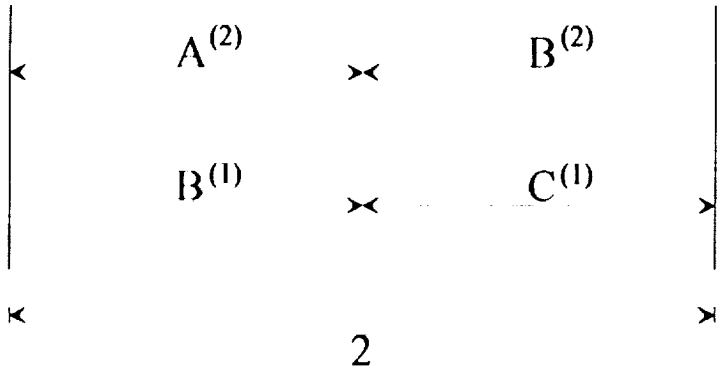
[2]See [1] or [3] for an alternate but equivalent definition.



(a)



(b)



(c)

Figure 1. (a) An AMG. (b) Execution of the AMG. (c) TGP diagram. Data packet numbers are shown in parenthesis.

2.2. *Parhi and Messerschmitt*

The terminology used in [2] corresponds quite closely to that of ATAMM. The counterpart to the ATAMM AMG is referred to simply as a *data flow graph*, or DFG. Like an AMG, a DFG consists of nodes and edges. However, there are no sources or sinks; every node is part of at least one *directed cycle*, which is defined in the usual graph-theoretical sense. A register is analogous to an initial token and specifies both initial data and delay. The "initial data" property is not explicitly stated in [2], but is implicit because a DFG must have at least one token in every loop to start up and reach steady-state. As [2] provides no notation for specifying "full" versus "empty" registers, it must be assumed that all registers are initially full.

One ATAMM data packet corresponds to one *iteration* of a DFG. TBO_{ALB} is referred to in [2] as the *iteration bound*, or T_∞, but the more general TBO_{LB} has no analog since [2] is not concerned with running with fewer processors than is required to achieve the iteration bound. A processor schedule in which the actual *iteration period* is equal to the iteration bound is said to be a *rate-optimal* schedule. *Iteration number* is equivalent to packet number.

Throughout the remainder of this paper terms from both ATAMM and [2] will be used as is appropriate.

3. Comparison of ATAMM Scheduling to Parhi and Messerschmitt Scheduling

3.1 *Fully Static Scheduling*

3.1.1. Classes of Schedules

Three terms which are used in [2] but not in the referenced ATAMM literature are *overlapped schedule*, *fully-static schedule*, and *cyclo-static schedule*. The following definitions are consistent with those presented in [2]. A schedule is overlapped if any node of packet[3] N+1 fires before all nodes of packet N have completed. The strategies used in ATAMM, [2], and [5] may create overlapped schedules. A periodic schedule is fully-static if all instantiations of any given node are scheduled on the same processor. A

[3]The phrase "node of packet N" is not strictly consistent with the definitions of node and packet; technically the phrase should be "node instantiation that produces a token of packet N" However, the former phrase will be used for brevity when the meaning is unambiguous.

periodic schedule is cyclo-static if the following condition is met. If any given node is instantiated on processor P_k for packet p then it is instantiated on processor $P_{(k+K) \bmod N}$ for packet p+1, where K is some integer constant and N is the number of processors. A schedule must be periodic if it is to be either fully- or cyclo-static. These terms are more rigorously defined in [2].

There exist classes of scheduling which impose constraints weaker than cyclo-static. *General periodic schedules* may or may not be cyclo-static. That is, the function which maps node iterations to processors may not be linear modulo the number of processors, as is necessary for a cyclo-static schedule. [2] and [5] are concerned only with compile-time schedules, whereas ATAMM scheduling is done at run-time, although scheduling performance is predicted at design time. With regard to the question of which processor executes which node iteration, run-time assignment of nodes to processors can be *unpredictable*[4]. For example, consider Figure 1(c). In the current ATAMM implementations, at the time instant t when both $A^{(2)}$ and $B^{(1)}$ complete, the two processors enter a race condition, the winner of which will run whichever of $B^{(2)}$ or $C^{(1)}$ has higher priority. The node priority is only used in node to processor assignment but all nodes are executed as soon as they are enabled. Thus a general periodic schedule will result and time performance and periodicity are not adversely affected.

3.1.2. Fully-Static Schedules

In [2], it is shown that for any DFG there exists a fully-static rate-optimal schedule, given adequate resources. However, it is crucial to note that Parhi and Messerschmitt achieve a schedule which is fully-static *with respect to the unrolled DFG, not the original DFG*, although curiously this fact is not directly stated in [2]. As far as the original DFG is concerned, the schedule that their unfolding-based algorithm creates is actually cyclo-static. For example, consider Figures 2 and 3(a), both taken from [2], which show a DFG and the resulting rate-optimal schedule. To see why Parhi and Messerschmitt did not specifically address this issue, note that under the assumption that nodes may be multiply instantiated (or equivalently for purposes of this argument, that the graph may be unfolded), it will *not* be possible to create a fully-static schedule if the time of the largest node is greater than TBO_{ALB}. Clearly any two or more distinct instantiations of the same node which are running at the same time instant must be on two distinct processors. In this paper, schedules that are fully-static except for violations caused by node execution times that are greater than TBO_{ALB} will be referred to as *optimally-static*. An optimally-static schedule is technically cyclo-static, but it is also as fully-static as it can be; hence the use of a new term. Also note that even if no node has execution time greater than TBO_{ALB}, to achieve a fully-static rate-optimal schedule may require more processors

[4]Historically, no effort was made to show that ATAMM architectures exhibited fully-static or cyclo-static behavior (although they are periodic), so ATAMM schedules might be placed in this category. However, in the future it is planned that ATAMM schedules may be made fully-static through the addition of processor/node constraints.

than for a cyclo-static rate-optimal schedule[5]. Again consider the example shown in Figures 2 and 3(a), which were taken from Figure 7 of [2]. The DFG is shown in Figure 2. Figure 3(a) repeats the schedule shown in [2]; this schedule is the minimal-processor rate-optimal schedule and is cyclo-static with respect to the graph of Figure 2. Note that this schedule is as static as it can be, given the 3 processor limitation. Figure 3(b) shows that in this case the addition of 1 extra processor allows for an optimally-static schedule.

Not only is the schedule in Figure 3(a) not fully-static with respect to the graph of Figure 2, it is not even periodic with respect to a period of length $TBO_{A,B}$; its actual period is three times longer than $TBO_{A,B}$. A modified version of the schedule that is periodic with period $TBO_{A,B}$ is shown in Figure 3(c).

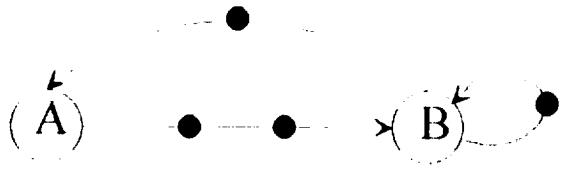
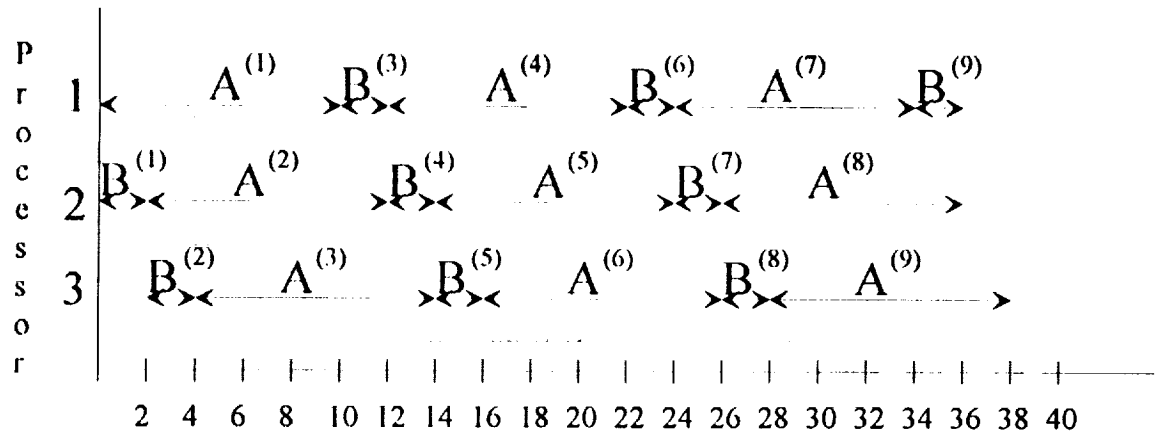
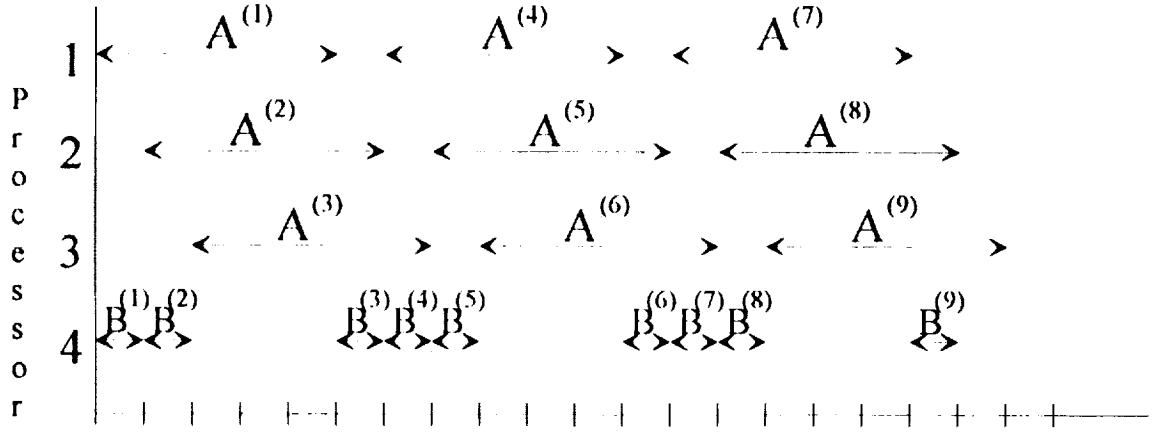


Figure 2. An AMG with initial tokens.

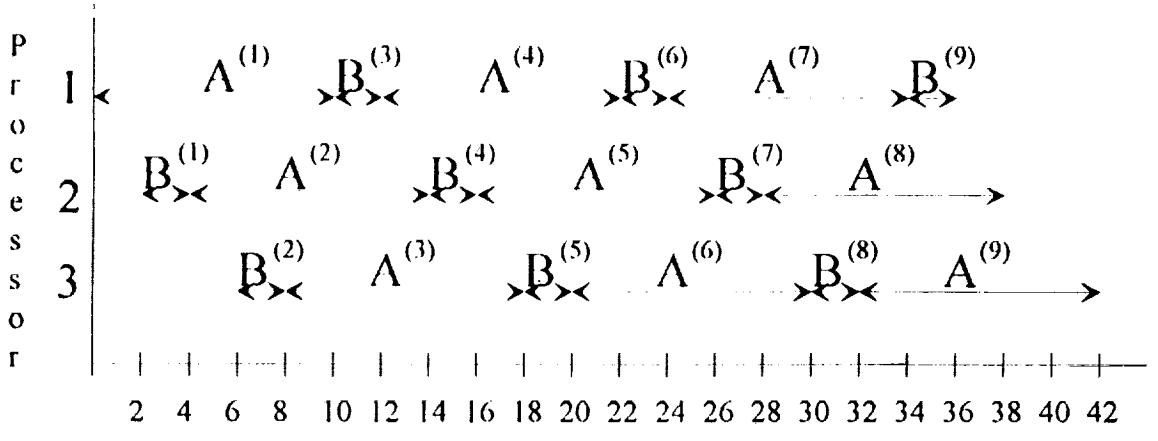
[5]This follows easily from the second point, column 2, p. 355, of [5]



(a)



(b)



(c)

Figure 3. (a) Cyclo-static schedule. (b) Optimally-static schedule. (c) Improved cyclo-static schedule with period equal to TBO_{ALB}.

The primary achievement of [2] is to show that any DFG can be scheduled rate-optimally in a fully-static manner with respect to the *unfolded* version of the DFG. Before effort is made to show that ATAMM schedules can be made to be optimally-static with respect to the *original* graph, it seems wise to consider the importance of optimally-static scheduling from an engineering point of view. Such an engineering viewpoint was not adopted in [2] due to the fact that determining a fully-static (or optimally-static) rate-optimal schedule is an NP-complete problem [6], as pointed out in [2].

Nonetheless, the optimally-static property is of practical as well as theoretical interest. If a node can be executed on any processor, as is the case in current ATAMM-based architectures, then every processor requires a copy of the code for that node. Such replication is a strength if a high degree of fault-tolerance is desired [3], but is wasteful if memory conservation is a major concern.

Both optimally-static and cyclo-static scheduling have a communication delay and bandwidth usage advantage over dynamic scheduling if the processors are connected via a non-bus (i.e. point-to-point) network. For a dynamic schedule, it can easily be the case that any node will run on many different processors, and potentially on all processors at different points in time. Thus, depending on the implementation of course, if a copy of each node[6] is kept on every processor on which it may run (which in general may be all processors), then graph updates such as making a token available on an incoming edge must be broadcast to all processors. This is not a problem for current ATAMM implementations because the communication network is always a bus.

If an optimally-static schedule is known, then of course the processor[7] on which each node will run is a known constant, in which case the potentially high overhead of broadcasting on a non-bus network is eliminated. In the case of a cyclo-static schedule, the processor for each node is by definition not necessarily a constant, but nonetheless the processor which will run the next instantiation of a node is known (if the schedule is determined before run-time) or can be computed on the fly (if dynamic scheduling is used). The destination processor for graph maintenance messages is known and broadcasting is not necessary.

3.2. Comparison of memory requirements

To achieve rate-optimal operation, there must be a mechanism for exploiting sufficient pipeline concurrency. In [2] this is achieved through overlapped schedules and

---

[6]A *copy of a node* might typically include the code for the node plus graph information such as lists of incoming and outgoing arcs, and buffer space for incoming tokens waiting to be used.

[7]Of course, if the computation time of a node is greater than TBO, then it is actually the *set* of processors on which the node will run that is known.

optimum unfolding. In ATAMM the mechanisms are overlapped schedules and multiple node instantiations.

Although memory requirements were not of concern in [2] due to its theoretical point of view, it is nonetheless interesting to compare the memory requirements of optimum unfolding to a dynamic scheduling system typified by ATAMM. One measure for this comparison is the peak amount of memory required for node instantiations; unit memory requirements for each node will be assumed. Another measure is the required number of buffers on the edges of a data flow graph.

In [2], nodes are not multiply instantiated so the node measure is easy to compute; it is simply JN, where J is the unfolding factor and N is the number of nodes in the DFG. From the unfolding algorithm it can also be seen that the number of edges and hence minimal number of buffers[8] is also JN. Thus from the memory point of view every node is multiply instantiated J times whether it "needs" to be or not, and every edge is buffered J times whether it "needs" to be or not. Whether or not a node needs to be multiply instantiated or multiply buffered is a function of its computation time and TBO.

Due to its use of injection control ATAMM will require no more than the minimum number of instantiations for each node, and relatively few buffers. First consider multiple node instantiations. Let A be a node and T(A) be the execution time of the node, then from [3]

$$\{\text{number of instantiations}\}(A) = \left\lceil \frac{T(A)}{TBO} \right\rceil \tag{3}$$

It is easy to see that $\left\lceil T(A)/TBO \right\rceil$ is a lower bound on the number of instantiations. In steady-state exactly one instance of A must complete every TBO interval. Each instance of A takes T(A) units of computing time to complete execution. Hence, {number of instantiations}(A) $\geq$ T(A)/TBO, and the lower bound follows immediately. The fact that $\left\lceil T(A)/TBO \right\rceil$ is also an upper bound is less obvious, but assuming that the graph is periodic in TBO[9], this bound follows from the fact that each instantiation of a node begins at the same time offset from the beginning of a TBO interval.

[8]Each edge requires at least one buffer to hold a token from the time it is generated by the predecessor node until the time when it is used by the successor node. It is possible to assume that the successor node maintains this buffer rather than the edge, but the buffer must exist somewhere.

[9]A proof that the ATAMM strategy leads to periodic execution is given in [1] for graphs which contain at most one token per edge and which are run without multiple node instantiations. The proof is presently being extended to show that graph plays remain periodic even when the above restrictions are lifted.

Turning to the number of buffers required for a single edge, the upper bound is given by

$$\textbf{number of buffers} \leq \left\lceil \frac{\mathbf{TT}}{\mathbf{TBO}} \right\rceil \tag{4}$$

where TT is the total packet computation time as defined in [1]. TT represents the maximum length of time the graph may take to fully process a packet. Thus just before a packet completes, $\lceil TT/TBO \rceil$ new packets may have started, and by research now in progress[10], Equation 4 is a valid upper bound.

Optimum unfolding will need to unfold the graph *at least*

$$J = \left\lceil \underset{\{\text{all nodes } A_i \text{ in DFG}\}}{\mathbf{Max}} T(A_i) / TBO \right\rceil \tag{5}$$

times, and most probably many more times than that. In the worst case the unfolding factor may be exponential in the number of arcs with registers[11]. Therefore J copies of each node will be made. J copies of each edge will also be made, and so J buffers will be required. Under ordinary circumstances we could expect the number of instantiations required by (3) and the number of buffers required by (4) to be considerably less than the number required by quantity (5), at least for most nodes/edges.

The DFG of Figure 2 provides an illustrative example of when optimum unfolding does and does not cause extra nodes and edges to be instantiated. From the schedule of Figure 3(a), node A must be instantiated 3 times concurrently, but B only needs to be singly instantiated. The optimally unfolded version of Figure 2 is given in Figure 4. This graph contains 3 instances of node A, 3 instances of node B, and three copies of each edge. All 3 instances of A are required, but only 1 instance at a time of B is needed.

Although this example is relatively kind to optimum unfolding in that an exponential unfolding factor is not required, it is still seen that optimum unfolding requires considerably more than the minimum amount of memory in terms of node instantiations and edge buffers.

---

[10]Private Communication with Robert L. Jones, NASA Langley Research Center, Hampton, VA, Summer 1992.

[11]Remark 7.2 of [2] implies this fact. Clearly, ordering the loops in an optimally unfolded DFG is not exponential in anything, unless the *number of loops or the number of nodes in some loop is exponential* with respect to some quantity. The latter possibility is the case. The worst-case number of nodes in a loop of the unfolded graph is linearly proportional to the unfolding factor J, which is equal to the least common multiple of the number of registers in all loops, a quantity which can be exponential in the number of loops (and thus edges) of the original graph.

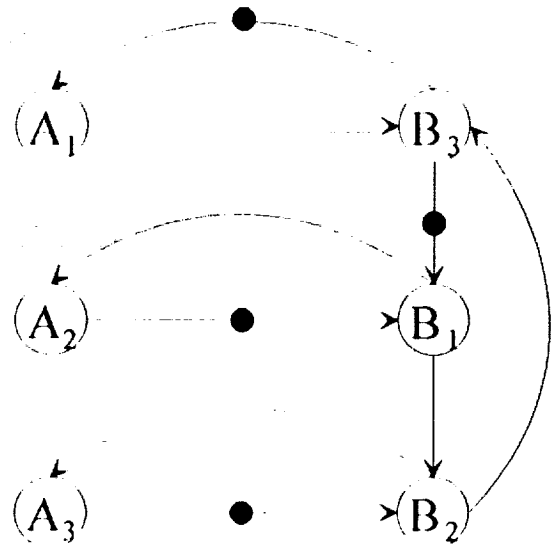


Figure 4. The graph of Figure 2 unfolded by a factor of 3.

4. Other Related Work

4.1. Range Chart Scheduling

An alternative method of scheduling DFGs is given in [5]. The primary tool used to determine a schedule in [5] is the *range chart*. A range chart is equivalent to a TGP diagram with the execution time of each node extended by the float time of that node. As a result, the point of view taken in [5] is very similar to that of ATAMM, which is based on the concept of TGP. Like ATAMM but unlike optimum unfolding as presented in [2], the range chart technique is applicable to data flow graphs which may or may not contain loops; there is no requirement that every node be in some loop. One difference between [5] and recent ATAMM work is that [5] assumes the longest node execution time to be an additional lower bound on TBO_{ALB}.

Two specific problems are tackled in [5], and both are solved with essentially the same heuristic. The first problem is to minimize the number of processors required to schedule a data flow graph, given a fixed TBO value greater than or equal to TBO_{ALB}', where the prime indicates the additional constraint of longest node execution time on TBO_{ALB}. The second problem is to find a fully-static schedule that minimizes TBO for P processors, where P is greater than or equal to the number required for a fully-static schedule at TBO_{ALB}'.

The heuristic for the processor minimization problem is presented first, and operates as follows. The range chart is computed and a node is chosen as the reference

node. The choice of reference node is a weak spot in the algorithm, as the performance of the heuristic is dependent on the choice of reference node, and the only recommendation that the authors make is that the reference node be "carefully chosen" [5]. The algorithm enters phase 1 and for each node does the following:

1. A pointer to the *current level* is maintained for each time unit in the TGP. Initially this pointer is set to 1 for all time units.

2. The node with least scheduling range is chosen for processing.

3. The start time of the node within the TGP is chosen so that the node runs within its scheduling range and occupies the lowest possible level at each time unit. Note that since levels do *not* necessarily correspond to processors in this step, a single node may occupy more than one level. The pointers are updated to reflect the new first available level of each time unit.

After phase 1 the algorithm sorts the nodes in decreasing order of execution time, and then enters phase 2, executing the following steps for each node:

4. A pointer to the current level is maintained for each time unit in the TGP. Initially this pointer is set to 1 for all time units. In this phase a level *will* correspond to a processor.

5. The first node from the sorted list is removed and assigned to the first level that has available all the time slots that the node requires. The pointers are updated for the time units now occupied by the node.

The algorithm for the TBO-minimization problem uses the above algorithm as a subroutine. The TBO-minimization algorithm operates as follows. The number of available processors P is given, and TBO is initially guessed to be TBO_{ALB}. The processor-minimization algorithm is run, and if in either phase the algorithm needs more than P levels, it is aborted and restarted with a new TBO guess one time unit longer than the previous guess. It is not explicitly mentioned in [5], but this linear search for the minimum feasible TBO could be changed to a binary search by making use of the knowledge that the number of nodes is an upper bound on the number of processors required (recall that multiple instantiation is not allowed in [5]). In this way the running time would be increased logarithmically rather than linearly from the time of the processor-minimization algorithm.

Given the beneficial properties of fully-static scheduling as discussed in Section 3.1.2, it may be desired to develop an optional additional node-binding step in the ATAMM design procedure for achieving fully-static (or cyclo-static if that is all that is

desired) operation. The node-binding step assumes a TGP with float times has been constructed[12]. Nodes can then be assigned to processors in one of three ways.

1. For sufficiently small graphs, an exhaustive search using branch-and-bound or other speedup technique(s) could be used.

2. A heuristic algorithm, such as the one presented in [5] could be used; this is perhaps the best alternative for large graphs. As pointed out in [5], two interdependent decisions need to be made in order to assign nodes to processors: the nodes need to be fixed in time, and bin-packed onto processors. The heuristic in [5] first makes the time decision for all nodes, then makes the processor decision for all nodes. While this seems to be a reasonably intuitive approach, there is no obvious reason why a different approach cannot be tried, such as fixing a node in time, assigning it to a processor, and repeating the procedure for each node. This "one node at a time" approach is likely to be the one a human would use if attempting to create the schedule by inspection, as in alternative 3 below.

   A heuristic algorithm will require the construction and maintenance of a range chart, which can be done either with existing ATAMM code that computes a TGP with node floats, or with the range chart construction algorithm provided in [5]. A comparison of the running time of these two methods would be interesting but is beyond the scope of this paper.

3. The user can be relied upon to assign nodes to processors by inspection. This procedure could be supported in the ATAMM Design Tool[10] through the addition of a resource display that allows nodes to be placed on processors one at a time in sequence. Alternatively, the user could be shown an initial display, such as one node per processor, and then be allowed to change the processor on which a node is running, as long as the resulting schedule is feasible. Node assignment should be an easy task as the float available to a node is dynamically updated in the Design Tool. As nodes are moved around, and if desired by the user, a node could be fixed in time thus eliminating its float. Currently, the ATAMM design procedure uses control edges as a mechanism for controlling placement of a node within its float. Control edges do not allow arbitrary placement of a node within its float time, but nonetheless in a bin packing situation the only meaningful times to start a node are upon completion of another node (also, in a data flow architecture the only event which can start a non-source node is the finish of some other node).

[12]Recall that such a TGP is equivalent to the scheduling-range chart of [5], so if the performance of the currently-used ATAMM algorithm for finding the TGP and floats ever proves unsatisfactory, a rough time complexity comparison could be made between it and the algorithm used to compute range charts in [5]. The range-chart algorithm would then offer an alternative if its performance was shown to be better.

4.2. Optimal Processor Assignment for Pipeline Computations

Another paper that address the scheduling of iterative data flow graphs is [7], but in this work a different point of view is taken. In ATAMM, [2], and [5], it was assumed that nodes may be neither split nor joined -- that the data flow graph expresses whatever concurrency is and is not available in the problem. In [7], a fundamentally different assumption is made that it is possible to speed up a node by allocating multiple processors to it, and that a function is known for each node, either through analysis or (more likely) through experimentation, that maps the number of available processors for the node to execution time for the node. In most cases, it is expected that assigning more processors to a node will generally decrease the node execution time, although it is possible that assigning more processors may actually increase execution time. A further assumption is that the input data flow graphs do not have initial tokens or recurrence loops.

The purpose of [7] is to set up and solve the mathematically well-defined *response time optimization* problem. Given the above assumptions. the problem is as follows. Given an upper bound on the total number of processors that may be used, and an upper allowable bound on the resulting TBO, choose an allocation of processors to nodes of the data flow graph such that the execution time (i.e. TBIO) of the graph is minimized.

[7] also defines an analogous *throughput optimization* problem. Again given an upper bound on the total number of processors available, and an upper bound on TBIO, the problem is to find an allocation of processors to nodes such that TBO is minimized.

Note the similarity of the above two problems to the TBO versus TBIO tradeoff in ATAMM operating points, but also note that in this case, unlike in ATAMM, the choice of "operating point" is a matter of how many processors are assigned to each node.

For highly-specific, precisely-stated mathematical optimization problems such as those in [7], it is dangerous to make comparisons with other problems since often a seemingly minor difference in problem statement or the assumptions can lead to quite a different problem. Nonetheless an attempt will be made to relate the throughput optimization problem to the ATAMM model. In the ATAMM model there is no way to speedup the execution time (TBIO) of a task by using additional processors, so it is very difficult to relate the response time problem to ATAMM. However, for the types of data flow graphs considered in [7], namely trees and series-parallel graphs, it *is always* possible (because these graphs have no loops) to increase throughput by allowing for additional multiple node instantiations. So, given the appropriate type of data flow graphs, it is possible to use the throughput optimization algorithm of [7] to compute the number of processors to assign to each node, i.e. the number of multiple instantiations allowed for each node. Before the algorithm can be used, it is necessary to determine the processor count to execution time mapping, but in the ATAMM model this function is always linear:

$$\text{Effective execution time} = \frac{\text{Node execution time}}{\text{Number of instantiations}}$$

This "execution time" is *not* TBIO. Instead, it is necessary to view n consecutive iterations of a node as a single "task", where n = Number of instantiations.

While a mapping between the problem domains of ATAMM and [7] can be made as above, it is of little practical value since the dynamic programming algorithm in [7] is designed to handle arbitrary processor to execution time functions, and so is an overkill in terms of both conceptual and time complexity for the special case of a linear execution time function as results from ATAMM throughput speedup by multiple instantiation.

5. Conclusion

The primary objective of [2] is to show that for any DFG it is possible to find a rate-optimal schedule that is fully-static with respect to the unfolded graph and cyclo-static with period $J \cdot TBO_{ALB}$ with respect to the original graph. Because it may be necessary to unfold a DFG an exponential number of times, the cyclo-static period may be quite large.

Algorithm 7.1 of [2], which runs in exponential time, constitutes the proof that a schedule with the desired properties can be found. As a side effect of this algorithm, an upper bound on the number of processors required to execute the schedule is found.

If (and only if) the number of processors used is not an issue, the determination of a rate-optimal schedule is trivial. For each node A, simply assign $P_A = \lceil T(A)/TBO_{ALB} \rceil$ processors to A and allow P_A concurrent instantiations of A. With such an assignment, the graph will never enter a resource-limited mode, and will execute at $TBO = TBO_{ALB}$ [1]. Furthermore the schedule will be optimally-static since a node is never executed on a processor not originally assigned to it, and no more processors are assigned to a node than are required by that node.

The difficult aspect of the problem is attempting to minimize the number of processors required for a optimally-static rate-optimal schedule. A nearly identical problem is discussed in [5], the difference being that multiple instantiations are not allowed since [5] uses the strict definition of fully-static; the problem is found to be NP-complete as expected. As is done in [5], it is wise to directly attack, either through heuristics or exhaustive search, the problem of optimizing the number of processors required to achieve desired execution properties, such as a rate-optimal TBO and optimally-static processor assignment.

6. Future Work

The following is a summary of recommendations for possible future work for the ATAMM project, some of which have already been mentioned in various places in this document.

6.1. Theoretical Work

- Formally prove that a graph executing under ATAMM rules will eventually reach steady-state.

- Formally prove an upper bound on the number of transients that occur from the time when a graph begins execution until steady-state is reached. (this is a stronger version of the previous item.)

- Since required memory space appears to be a major advantage of ATAMM *vis a vis* optimum unfolding, additional analysis, both theoretical and empirical examples, could be used to prove this point.

6.2. Implementation Work

- Develop an optional additional node-binding step in the design procedure for achieving fully-static (or cyclo-static if that is all that is desired) operation, as discussed in Section 4.1.

- Evaluate the node to processor assignment strategy by simulation.

References

[1] S. Som, J. W. Stoughton, and R. R. Mielke, "Strategies for Concurrent Processing of Complex Algorithms in Data Driven Architectures," NASA Contractor Report 187450, Grant NAG1-683, October 1990.

[2] K. K. Parhi and D. G. Messerschmitt, "Static Rate-Optimal Scheduling of Iterative Data-Flow Programs via Optimum Unfolding," IEEE Trans. Computers, vol. 40, pp. 178-195, February 1991.

[3] R. Mielke, J. Stoughton, S. Som, R. Obando, M. Malekpour, and B. Mandala, "Algorithm to Architecture Mapping Model (ATAMM) Multicomputer Operating System Functional Specification," NASA Contractor Report 4339, Cooperative Agreement NCC1-136, November 1990.

[4] R. Mielke, J. Stoughton, S. Som, "Modeling and Optimum Time Performance for Concurrent Processing," NASA Contractor Report 4167, Grant NAG1-683, August 1988.

[5] Sonia M. Heemstra de Groot, Sabih H. Gerez, Otto E. Herrmann, "Range-Chart-Guided Iterative Data-Flow Graph Scheduling," IEEE Trans. Circuits and Systems-I, vol.39, no. 5, pp. 351-364, May 1992.

[6] M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of the NP-Completeness, San Francisco, CA: Freeman, 1979.

[7] D. M. Nicol, R. Sinha, A. N. Choudhury, B. Narahari, "Optimal Processor Assignment for Pipeline Computations", NASA Contractor Report 189550, Contract No. NAS1-18605, October 1991.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 1993	Contractor Report 6/29-8/14/92

4. TITLE AND SUBTITLE
A Comparison of Multiprocessor Scheduling Methods for Iterative Data Flow Architectures

5. FUNDING NUMBERS
G NAG1-613

WU 586-03-11

6. AUTHOR(S)
Matthew Storch

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Illinois
The Department of Computer Science
Illinois Computing Laboratory for Aerospace Systems and Software (ICLASS)
Urbana, Illinois 61801

8. PERFORMING ORGANIZATION REPORT NUMBER
UIUCDCS-R-92-1776
UILU-ENG-92-1756

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CR-189730

11. SUPPLEMENTARY NOTES
Langley Technical Monitor of Grant: Kathryn A. Smith
Langley Technical Monitor for this supplemental research: Paul J. Hayes

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 33

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

A comparative study is made between the Algorithm to Architecture Mapping Model (ATAMM) and three other related multiprocessing models from the published literature. The primary focus of all four models is the non-preemptive scheduling of large-grain iterative data flow graphs as required in real-time systems, control applications, signal processing, and pipelined computations. Important characteristics of the models such as Injection Control, Dynamic Assignment, Multiple Node Instantiations, Static Optimum Unfolding, Range-Chart Guided Scheduling, and Mathematical Optimization are identified. The models from the literature are compared with the ATAMM for performance, scheduling methods, memory requirements, and complexity of scheduling and design procedure.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Iterative data flow, multiprocessing, static scheduling, dynamic scheduling	21
	16. PRICE CODE: A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL